Ontopy : programmation orientée ontologie en Python⋆

Jean-Baptiste Lamy[1], Hélène Berthelot[1]

LIMICS, Université Paris 13, Sorbonne Paris Cité, 93017 Bobigny, France, INSERM UMRS 1142, UPMC Université Paris 6, Sorbonne Universités, Paris
jean-baptiste.lamy@univ-paris13.fr, helene.berthelot@orange.fr

Résumé : Les ontologies et les modèles objets partagent un vocabulaire commun mais diffèrent dans leurs utilisations : l'ontologie permet d'effectuer des inférences et les modèles objets sont utilisés pour la programmation. Il est souvent nécessaire d'interfacer ontologie et programme objet. Plusieurs approches ont été proposées, de OWL API à la programmation orientée ontologie. Dans cet article, nous présentons Ontopy, un module de programmation orientée ontologie dynamique en Python, et nous prendrons pour exemple la comparaison des contre-indications des médicaments.
Mots-clés : Ontologies, Programmation orientée ontologie, Programmation dynamique

1 Introduction

Les ontologies formelles, par exemple au format OWL (*Ontology Web Language*), structurent un domaine de connaissance pour réaliser des inférences logiques et relier les connaissances entre elles. Des éditeurs comme Protégé rendent facile la construction d'ontologies, mais leur intégration à des logiciels existants est plus compliquée (Goldman NM, 2003). Il existe des similitudes entre ontologie et modèle objet (Koide *et al.*, 2005) : les classes, propriétés et individus des ontologies correspondent aux classes, attributs et instances des modèles objets (Knublauch *et al.*, 2006). Cependant, les principaux outils comme OWL API (Horridge & Bechhofer, 2011) n'en tirent pas parti : avec ces outils une classe de l'ontologie *ne correspond pas* à une classe du langage de programmation. Ces outils sont par conséquents complexes à mettre en oeuvre et difficilement compatibles avec les méthodes de développement agile. Une approche différente consisterait à aller vers le rapprochement, voire l'unification, des ontologies et des modèles objets : c'est la *programmation orientée ontologie* (Goldman NM, 2003). Sur un exemple du W3C, cette approche a permis de réduire de moitié le volume de code source (Knublauch *et al.*, 2006).

Cet article présente Ontopy, un module Python pour la programmation orientée ontologie dynamique. Ontopy permet de créer et manipuler les classes et les instances OWL comme des objets Python, et de classifier automatiquement des classes et des instances *via* un raisonneur externe. Nous présentons ensuite le problème de la comparaison des contre-indications des médicaments, que nous réalisons avec une ontologie et un programme objet. Nous montrerons un exemple d'utilisation d'Ontopy dans ce contexte. OWL API n'a pas été utilisé car peu adapté à nos méthodes de développement agile, de plus nous souhaitions réutiliser des outils terminologiques mis au point précédemment en Python (Lamy *et al.*, 2015). Nous terminerons en comparant notre approche à la littérature.

⋆. Ce travail a été financé par l'ANSM au travers du projet de recherche VIIIP (AAP-2012-013).

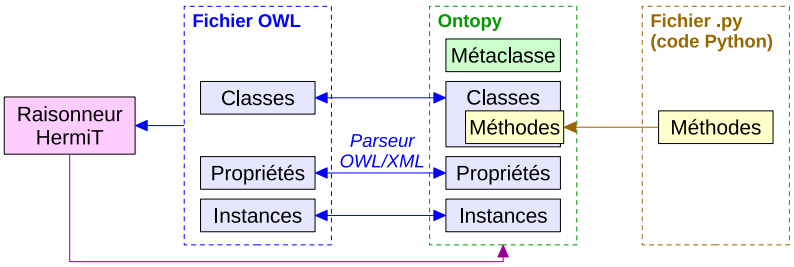

FIGURE 1 – Architecture générale d'Ontopy.

2 Ontopy : un module Python pour la programmation orientée ontologie

Ontopy est un module Python en logiciel libre (licence GNU LGPL v3, `https://bitbucket.org/jibalamy/ontopy`) pour la programmation orientée ontologie et le développement agile d'application à base d'ontologie. Le langage Python 3.4 a été choisi car il s'agit d'un langage objet dynamique avec héritage multiple. En particulier, il permet de changer la classe d'un objet ou les superclasses d'une classe en cours d'exécution, par exemple suite à la classification, ce que ne permet pas un langage statique comme Java. Ontopy permet (a) de charger des ontologies au format OWL 2 XML, (b) d'accéder au contenu de l'ontologie comme s'il s'agissait d'objets Python, (c) de créer des classes OWL en Python, (d) d'ajouter des méthodes Python aux classes OWL, et (e) d'effectuer la classification automatique des instances, classes et propriétés. Les types de données suivants sont gérés : booléen, entier, flottant, date, chaîne de caractères.

Ontopy (Figure 1) n'effectue aucune inférence (hors mise à jour des propriétés inverses) tant que le raisonneur n'est pas appelé explicitement. Ce comportement est similaire à celui de Protégé. Nous avons utilisé le raisonneur HermiT 1.3.8 (Motik *et al.*, 2009) auquel nous avons ajouté une option en ligne de commande pour obtenir en sortie la classification des instances. La classification se fait en 3 étapes : (1) exporter l'ontologie dans un fichier OWL temporaire, (2) exécuter HermiT sur ce fichier, (3) récupérer la sortie d'HermiT et appliquer les résultats en changeant les classes des instances et les superclasses des classes.

Python permet de modifier son modèle objet via un système de *métaclasses* (classe de classe). La Table 1 montre les méthodes spéciales que nous avons redéfinies pour adapter le modèle objet Python à OWL. Deux autres différences ont demandé un traitement particulier : (1) dans une ontologie, une instance peut appartenir à plusieurs classes, ce que ne permettent pas les langages objets ; dans ce cas, une classe intersection héritant des différentes classes est créée automatiquement et associée à l'objet, (2) les annotations ne sont pas héritées dans les ontologies, alors que tous les attributs le sont dans les langages objets ; c'est pourquoi nous avons placé les annotations dans un dictionnaire à part qui fait correspondre une entité (ou un triplet) à un second dictionnaire, lequel fait correspondre les propriétés d'annotation à leurs valeurs.

3 Le problème de la comparaison des contre-indications des médicaments

Le processus complexe de rédaction, structuration et codage des propriétés des médicaments conduit à une grande hétérogénéité dans les bases de données, qui com-

Méthode	Effet	Raison de la redéfinition
C.__new__	Crée un nouvel objet	Combiner la nouvelle classe à la classe OWL de même nom, si elle existe
C.__instancecheck__	Teste si un objet est une instance de la classe	Prendre en compte les classes équivalentes OWL
C.__subclasscheck__	Teste si une classe est une sous-classe de la classe	Prendre en compte les classes équivalentes OWL
C.mro	Calcule l'ordre de résolution des méthodes (*method resolution order*, MRO) notamment en cas d'héritage multiple	Ne pas déclencher d'erreur en cas de MRO temporairement incorrect lors du chargement de l'ontologie (les classes parentes étant ajoutées une à une)
i.__setattr__	Modifie un attribut de l'objet	Mettre à jour les propriétés inverses
i.__getattr__	Obtient un attribut de l'objet (appelé uniquement pour les attributs inexistants)	Retourner une liste vide si la propriété n'a pas été renseignée, ou None pour une propriété fonctionnelle

TABLE 1 – Méthodes spéciales du modèle objet de Python qui ont été redéfinies pour le rendre compatible avec OWL. Pour chaque méthode est indiqué si elle s'applique aux classes (C.) ou aux instances (i.), son effet et la raison de sa redéfinition.

Condition clinique	ticagrélor	aspirine	héparine
maladie hémorragique	CI		
maladie hémorragique acquise		CI	
maladie hémorragique constitutionnelle		CI	CI

Condition clinique	ticagrélor	aspirine	héparine
maladie hémorragique	CI	CI	CI/ok
maladie hémorragique acquise	CI	CI	ok
maladie hémorragique constitutionnelle	CI	CI	CI

TABLE 2 – Trois contre-indications pour trois médicaments, issues de la base médicament Thériaque en haut, et telles qu'interprétées par un expert en bas (CI : contre-indiqué, ok : absence de contre-indication, CI/ok : contre-indiqué dans certaines situations seulement).

plique la comparaison entre médicaments. La Table 2 (haut) montre trois exemples de situations de contre-indication pour trois médicaments, extraits de la base Thériaque (http://theriaque.org). Cependant, bien que cela n'apparaisse pas dans ce tableau, le ticagrélor est contre-indiqué avec les maladies hémorragiques acquises et constitutionnelles, car contre-indiqué dans l'ensemble des maladies hémorragiques (héritage). Et l'aspirine est contre-indiquée dans les maladies hémorragiques car contre-indiquée à la fois dans celles acquises et constitutionnelles (partition). Enfin, il est possible de déduire les situations dans lesquelles un médicament *n'est pas* contre-indiqué, par exemple les maladies hémorragiques acquises pour l'héparine (à ne pas confondre avec l'absence de mention de contre-indication dans la base). La Table 2 (bas) montre l'interprétation que ferait un expert ; nous souhaitons automatiser ce raisonnement.

Nous avons structuré les contre-indications à l'aide d'une ontologie formelle, dans laquelle les conditions cliniques associées aux contre-indications sont décrites par un code dans une terminologie et un ou plusieurs qualifieurs tels que "acquise", "constitutionnelle", "antécédent",... Ces conditions cliniques sont représentées par des classes et non des instances, afin de pouvoir prendre en compte les relations *est-un* existant entre conditions cliniques (par exemple maladie hémorragique acquise *est une* maladie hémorragique).

4 Exemple d'utilisation d'Ontopy

Nous donnons ici un exemple d'application d'Ontopy au problème de la comparaison des contre-indications. Ontopy charge les ontologies à partir des répertoires locaux définis dans la variable globale `onto_path`, ou à défaut à partir de leur URL. `onto_path` se comporte comme le `classpath` de Java ou le `pythonpath` de Python, mais pour les fichiers OWL.

```
from ontopy import *
onto_path.append("/chemin/local/des/ontos")
onto_ci = get_ontology("http://test.org/onto_ci.owl").load()
#charge /chemin/local/des/ontos/onto_ci.owl ou http://test.org/onto_ci.owl
```

L'ontologie peut ensuite être utilisée comme un module Python, et la notation pointée usuelle permet d'accéder aux éléments de l'ontologie. Des attributs (`imported_ontologies`, `classes`, `properties`, etc) permettent de récupérer la liste des éléments d'un type donné.

```
onto_ci.Médicament # La classe http://test.org/onto_ci.owl#Médicament
```

Les classes de l'ontologie peuvent être instanciées en Python. La notation pointée permet d'accéder aux relations des instances. Les relations fonctionnelles ont une valeur unique, les autres sont des listes.

```
aspirine = onto_ci.Médicament("aspirine") # onto_ci.owl#aspirine
aspirine.noms_de_marque = ["Aspirine du Rhône", "Aspirine UPSA"]
```

Il est possible de créer des classes OWL en Python, en héritant de `Thing` ou d'une classe fille. Les attributs `is_a` et `equivalent_to` sont des listes correspondant aux superclasses et aux classes équivalentes OWL. Ces listes peuvent contenir des classes, mais aussi des restrictions portant sur une propriété (définies de manière similaire à Protégé), des énumérations d'instances (*one of*), ou plusieurs de ces éléments reliés par des opérateurs logiques ET (&), OU (|) ou NON (NOT). Les classes présentes dans `is_a` sont ajoutées aux superclasses Python, en revanche les autres éléments ne sont pas traités comme des classes par Ontopy. L'exemple ci-dessous crée la classe des maladies hémorragiques acquises, fille de `Condition_clinique`, et définie comme équivalente à une condition clinique associée au terme "maladie hémorragique" et ayant pour qualifieur Acquis.

```
class Maladie_hémorragique_acquise(onto_ci.Condition_clinique):
    equivalent_to = [ onto_ci.Condition_clinique
    & onto_ci.a_pour_terme     (SOME, onto_ci.Terme_maladie_hémorragique)
    & onto_ci.a_pour_qualifieur(SOME, onto_ci.Acquis) ]
```

Nous pouvons ensuite créer la première contre-indication et la relier à l'aspirine.

```
ci1 = onto_ci.Contre_indication()
aspirine.a_pour_contre_indication.append(ci1)
```

Relier cette contre-indication aux maladies hémorragiques acquises est un peu plus compliqué, car il s'agit d'une classe et non d'une instance. Pour cela nous modifions les attributs `is_a` de la classe `Maladie_hémorragique_acquise` et de l'instance `ci1`. L'attribut `is_a` d'une instance fonctionne de manière similaire à celui d'une classe, mais contient les classes auxquels appartient l'instance. Ci-dessous, nous spécifions que la contre-indication est reliée seulement à des maladies hémorragiques acquises, et que la classe des maladies hémorragiques acquises est reliée à notre contre-indication.

```
ci1.is_a.append(
    onto_ci.a_pour_condition_clinique(ONLY, Maladie_hémorragique_acquise) )
Maladie_hémorragique_acquise.is_a.append(
    onto_ci.est_condition_clinique_de(VALUE, ci1) )
```

Créons ensuite la classe définie des conditions cliniques contre-indiquées avec l'aspirine.

```
class Condition_CI_avec_aspirine(onto_ci.Condition_clinique):
    equivalent_to = [ onto_ci.Condition_clinique
    & onto_ci.est_condition_clinique_de(SOME, onto_ci.Contre_indication
        & onto_ci.est_contre_indication_de(VALUE, aspirine) ) ]
```

Ontopy permet aussi l'ajout de méthodes Python aux classes OWL, en redéfinissant les classes dans un module Python. Ce module peut être lié à l'ontologie via une annotation, de sorte à être chargé automatiquement avec l'ontologie. L'exemple suivant ajoute une méthode `teste_ci` à la classe `Médicament`. Elle prend en paramètre une classe de condition clinique et retourne une chaîne de caractères. La méthode récupère la classe des conditions cliniques contre-indiquées avec le médicament, en se basant sur son nom, et teste si la condition clinique est une classe fille avec l'opérateur `issubclass` de Python. Puis nous lançons le raisonneur et nous affichons les résultats.

```
class Médicament(Thing):
    def teste_ci(self, Condition):
        Condition_CI = onto_ci["Condition_CI_avec_" + self.name]
        if issubclass(Condition, Condition_CI): return "CI"
        [...] # XXX tester si le médicament est OK

onto_ci.sync_reasoner() # Lance HermiT et effectue la classification
print(aspirine.teste_ci(Maladie_hémorragique)) # => "CI"
```

5 Discussion et conclusion

La programmation orientée ontologie n'est pas une idée nouvelle et le W3C a déjà suggéré l'intégration de méthodes dans des classes OWL (Knublauch *et al.*, 2006). Des approches statiques ont été proposées (Kalyanpur *et al.*, 2004; Goldman NM, 2003), qui génèrent le code source de classes Java ou C# correspondant à une ontologie en OWL. Ces approches permettent d'accéder à l'ontologie et de vérifier le typage à la compilation, mais leur nature statique n'est pas adaptée à la classification automatique. Plus récemment, une approche semi-dynamique en Java (Stevenson & Dobson, 2011) a permis la classification des instances mais pas celle des classes. Une approche dynamique a été proposée en Common Lisp (Koide *et al.*, 2005), en utilisant un algorithme de subsomption spécifique pour l'inférence et non un raisonneur externe. Un prototype en Python a aussi été réalisé (Babik & Hluchy, 2006), mais ne va pas jusqu'à une syntaxe "entièrement Python" pour

définir les restrictions ou les relations. Une troisième approche consiste à concevoir de nouveaux langages, tel que Go ! (Clark & McCabe, 2006).

Au final, peu d'approches sont allées aussi loin dans l'unification entre modèle objet et ontologie que la nôtre. Ontopy n'a pas été optimisé en terme de performance car nous n'en avons pas ressenti le besoin : le temps consommé par la manipulation de l'ontologie en Python reste négligeable comparé au temps de raisonnement. La totalité de l'ontologie est chargée en mémoire, ce qui peut poser problème sur des ontologies volumineuses. Nous avons cependant réussi à charger IDOSCHISTO, une ontologie complexe sur la schistosomiase (Camara *et al.*, 2014). Une autre limite d'Ontopy est la prise en compte d'espaces de nom multiples et d'assertions présentes dans une ontologie mais portant sur des éléments d'une autre ontologie, qui enfreignent le principe d'*encapsulation* des langages objets (l'ensemble des informations d'un objet sont placées dans une seule "capsule").

Les perspectives de développement d'Ontopy incluent (a) la liaison à un *triple store*, afin de ne pas charger la totalité des ontologies en mémoire, (b) la traçabilité de l'ontologie d'origine de chaque assertion, afin de faciliter l'emploi d'ontologies modulaires, ainsi que (c) la génération automatique de boîtes de dialogue pour éditer les instances.

Références

Babik M. & Hluchy L. (2006). Deep Integration of Python with Web Ontology Language. In *Proceedings of the 2nd workshop on scripting for the semantic web*, Budva, Montenegro.

Camara G., Despres S. & Lo M. (2014). IDOSCHISTO : une extension de l'ontologie noyau des maladies infectieuses (IDO-Core) pour la schistosomiase. In *Actes du congrès d'Ingénierie des Connaissances (IC2014)*, p. 39–50, Clermont-Ferrand, France.

Clark K. L. & McCabe F. G. (2006). Ontology oriented programming in Go. *Applied Intelligence*, **24**, 3–37.

Goldman NM (2003). Ontology-oriented programming : static typing for the inconsistent programmer. In *Lecture notes in computer science : the SemanticWeb, ISWC*, volume 2870, p. 850–865.

Horridge M. & Bechhofer S. (2011). The OWL API : A Java API for OWL ontologies. *Semantic Web 2*, p. 11–21.

Kalyanpur A., Pastor D., Battle S. & Padget J. (2004). Automatic mapping of OWL ontologies into Java. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2004)*, p. 98–103.

Knublauch H., Oberle D., Tetlow P. & Wallace E. (2006). A Semantic Web Primer for Object-Oriented Software Developers. *W3C Working Group Note*.

Koide S., Aasman J. & Haflich S. (2005). OWL vs. Object Oriented Programming. In *the 4th International Semantic Web Conference (ISWC 2005), Workshop on Semantic Web Enabled Software Engineering (SWESE)*.

Lamy J. B., Venot A. & Duclos C. (2015). PyMedTermino : an open-source generic API for advanced terminology services. *Stud Health Technol Inform*.

Motik B., Shearer R. & Horrocks I. (2009). Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*, **36**, 165–228.

Stevenson G. & Dobson S. (2011). Sapphire : Generating Java Runtime Artefacts from OWL Ontologies. In *Lecture Notes in Business Information Processing, Advanced Information Systems Engineering Workshops* , volume 83, p. 425–436.